LA-UR-15-20503
Approved for public release; distribution is unlimited.

Title: Code Disentanglement: Initial Plan

Author(s): Wohlbier, John Greaton
Kelley, Timothy M.
Rockefeller, Gabriel M.
Calef, Matthew Thomas

Intended for: Report

Issued: 2015-01-27

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer,is operated by the Los Alamos National Security, LLC for the National NuclearSecurity Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Departmentof Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

research note

Computer, Computational, and Statistical Sciences Division

CCS–7: Applied Computer Science, CCS–2: Computational Physics & Methods

To/MS:	Distribution
From/MS:	John Wohlbier (CCS–2)
	Tim Kelley (CCS–7)
	Gabriel Rockefeller (CCS–2)
	Matt Calef (CCS–2)
Phone/FAX:	(7-3965)
Symbol:	CCS–7–036
Date:	April 23, 2014

Subject: Code Disentanglement: Initial Plan
Revision: 1.0

1 Overview

The first step to making more ambitious changes in the EAP code base is to disentangle the code into a set of independent, levelized packages. We define a *package* as a collection of code, most often across a set of files, that provides a defined set of functionality; a package a) can be built and tested as an entity and b) fits within an overall levelization design. Each package contributes one or more libraries, or an application that uses the other libraries. A package set is levelized if the relationships between packages form a directed, acyclic graph and each package uses only packages at lower levels of the diagram (in Fortran this relationship is often describable by the `use` relationship between modules). Independent packages permit independent—and therefore parallel—development. The packages form separable units for the purposes of development and testing. This is a proven path for enabling finer–grained changes to a complex code.

EAP-FA Plan

I. Restructure code
   - A. Factor into N packages
   - B. Develop package interfaces

   this effort

II. Clean up individual packages
   - A. Cut redundant code
   - B. Organize code to reflect logical structure

III. Migrate to target software architecture

Figure 1: How this effort fits into the larger EAP–FA plan.

In this memo, we use *physical independence* to refer to code that has been partitioned into packages. Physical independence does not imply logical completeness. For example, we expect that we can form separate Hydro and EOS packages. This does not mean that a correct hydrodynamics simulation can be performed without reference to an equation of state. Rather, it means that a higher–level component will coordinate calls to each package, instead of packages mutually calling one another.

This memo outlines a plan for transforming the EAP code base into a levelized set of packages with well–defined interfaces. The relationship between the work described here and the larger EAP–FA effort is shown in Fig. 1: this undertaking is only the first step. Subsequent efforts will clean up within individual packages and then migrate code to the target architecture—a software architecture that responds to future compute architectures.

The criterion for determining whether a package has been successfully transformed is that it can be built and tested independently of other packages at the same or higher levels in the levelization diagram. This

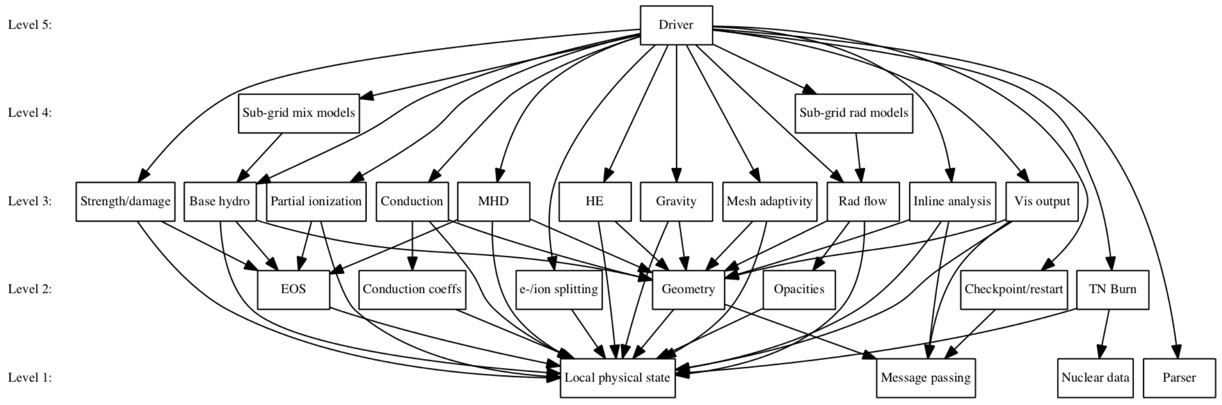


Figure 2: A very preliminary target level diagram for EAP codes.

ensures that the package does not participate in circular dependencies. When we arrive at the point at which the set of packages is levelized, we expect to be able to build the application with a simple link line—one in which no library appears twice.

We view complete packagization of xRage as a very substantial task (see, for example, Fig. 4, p. 3). We propose to begin with the existing directory structure, examining directory-level dependencies and transforming from intertwined directories to actual packages. We will migrate to the set of packages indicated by the EAP Physical Model, which is described in part by the (notional) levelization diagram (Fig. 2).[1]

We plan to begin by transforming one package whose dependencies we have already studied. The experience of packagizing one capability will inform our time estimates for packagizing the rest of the code. We believe an estimate of the amount of work required to transform the whole code will be much more accurate after the work proposed in this document has been completed, i.e., once we have empirical evidence of how much work it takes to clean up one package.

The recipe for packagizing a directory is summarized below:

1. Identify the dependencies between the selected package and other packages, using butterfly diagrams as described in § 2.

2. For each circular dependence, decide whether to eliminate the connection completely by deprecating the code in question.

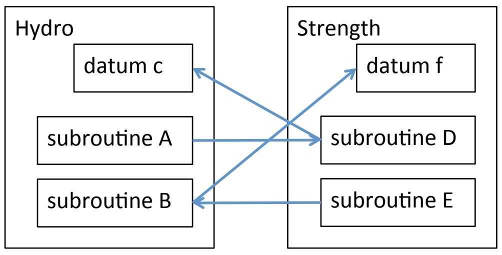
3. If a circular connection is not eliminated, make the connection one directional. We discuss criteria for this decision below and in § 5, but steps include checking test coverage of the code in question and examining the physical and logical coupling between the packages.

Once a decision is made about how to change a connection between packages, the package needs to be transformed. A set of transformations that achieve this levelization may be summarized as "out, split, and up":

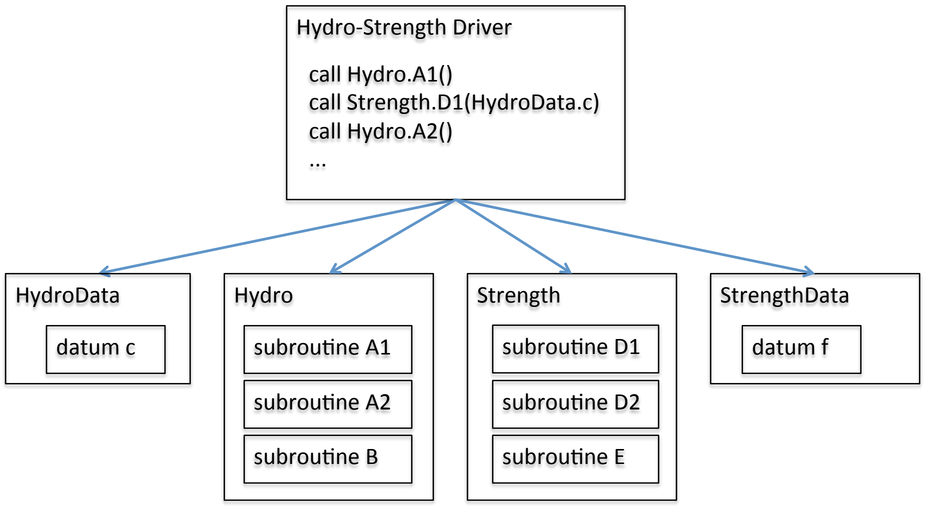
1. Move data and code that is used by multiple packages into a separate package (out).

2. Split a subroutine in one package A that calls code from another package B into two subroutines bracketing the call (split).

[1]Joann Campbell has pointed out that the Physical Model should not be confused with the Physics Model. While physics code (as well as non–physics code) is decomposed into packages according to the Physical Model, the Physics Model describes the relationships between different physics, including necessary ordering. The Physics Model describes protocols, while the Physical Model describes only how code is separated into packages.



(a)



(b)

Figure 3: Resolving circular package dependencies. (a) A typical circular dependency pattern between packages. (b) The same code after applying the "out" and "up" transformations. Data used by multiple packages have been moved out into separate packages. Code calls into separate packages are moved up into a common driver package that expresses the logical interleaving of the packages, while maintaining the physical decoupling. Note the absence of any lines from Hydro to HydroData: data required by subroutines in Hydro is passed in as explicit arguments by the driver, not via `use` statements. Of course, other packages may need to be integrated into this driver as well.

3. Move calls in package A to code from package B into a separate driver package (up). Replace package A's `use` of data from package B with explicit parameters in package A's interface.

These steps are illustrated in Fig. 3, p. 3. Many developers over the history of the project have pointed out the utility of this transformation and the resulting package structure.

A key aspect of the Rage architecture that must be accounted for is the *extras–chaining* mechanism. This mechanism is central to the application architecture for Rage. Essentially, extras–chains give the application a flexible mechanism to respond to runtime configuration information, such as which physics a user wants to use. Each chain also expresses an ordering between packages at a specific point in the calculation (whether or not that ordering is necessary). Extras–chains are involved in initialization, cycle execution, and finalization. As part of this initial effort, we will attempt to raise the extras–chaining to the driver levels. We will also examine alternatives that support a simpler method of accomodating runtime configuration.

Note that this process described so far captures only circular dependencies of the form `A-B-A`; and does not reveal dependencies of the form `A-B-C-A` and so on. Such longer loops do exist—e.g., in the extras–chains—but the larger process of moving the overall code into better agreement with the Physical Model will necessarily unravel those dependencies.

This is our first version of a method for detangling code packages. We will apply this to one package—Hydro—to begin with, observing any changes that need to be made. The result of our initial effort will be:

1. A Hydro package with no `use` statements for packages at the same level or higher;
2. A clear interface that expresses what any xRage hydro implementation must provide;
3. A method that we can repeat with subsequent packages.

As we gain confidence with the method, we will begin to work in parallel on a second package (Radiation), and we will confirm that the method is sound or make any necessary adjustments. At that point the method will be ready for wider–scale application, and we will engage the full EAP–FA team in this effort.

Once a directory has been successfully packagized and package interfaces have been defined, additional clean–up work can commence. Code within the package can be transformed subject to the contract of the upward interface. For example, we can define requirements for interfaces to lower packages and mitigate code redundancy.

The rest of this memo is organized as follows:

- § 2 describes how we selected and prioritized the first packages for packagization. In particular, we will begin with the Hydro package.
- § 3 describes how we analyzed the dependencies between Hydro and other packages.
- § 4 describes how we prioritize which changes within Hydro to make.
- § 5 describes code transformations in greater detail.
- § 6 lists completion criteria and what we expect to immediately follow the completion and dissemination of this memo.

2 Package Selection

xRage does not currently have *packages* as defined in § 1, rather it has directories containing files that have commonality in function. In particular the directories can not be built and tested as entities, nor were they created as part of a levelization design. The process described here moves from present directories to future packages.

Before diving into this cleanup activity we wanted to carefully choose the first set of directories to address. This section describes how we chose the Hydro directory as our first candidate, and the directories that would immediately follow the cleanup of Hydro. We set the following criteria to help us make our choice:

1. The directory should represent a physics package, and not some auxiliary package like checkpoint reading and writing or an input parser.
2. The physics package should be commonly used and not a less frequently used physics capability like self-gravity.
3. The Subject Matter Expert (SME) for the physics capability should be interested in and willing to help with our activity.
4. The physics package should be well covered by tests.
5. The complexity of the interdependencies of the chosen package with other packages should be non-trivial, where metrics for complexity will be described below.
6. The complexity of the interdependencies of the chosen package with other packages should not be overwhelmingly difficult.
7. The team members should have some level of familiarity with the physics package so as to eliminate complete reliance on SMEs.

The initial candidate directories were (in alphabetical order):

- EOS (equation of state)
- HEBurn (high explosives)
- Hydro
- Iso (isotopics)
- Plasma
- Radiation (radiation diffusion)
- Strength
- TNBurn (thermonuclear burn)
- Turbulence

Note that the mesh is not included in the list of initial candidates for packagization. While the mesh is certainly a candidate to be made into a package, executing the overall packagization process in a "top-down" fashion, starting with conventionally-recognized physics packages, allows us to catalog usage patterns of lower-level packages like "mesh" and prepares us to define interfaces to those packages.

We relied on Understand to measure complexity of directory interdependencies. Understand provides source level analysis and creates dependency graphs between directories and source files. The directory level dependencies for `Source.rh` are shown in Fig. 4. Understand draws a blue line connecting directories where at least one file in one of the directories depends on at least one file in the other directory. The arrow points to the "depended on" directory. Red lines are drawn when at least one file in the first directory depends on at least one file in the second directory, and at least one file in the second directory depends on at least one file in the first directory. This is what we refer to as a circular directory level dependency.

A more useful diagram for any one directory is a *butterfly dependency graph*. The butterfly dependency graph simply takes the directory in question and draws the directories that it is connected to. The lines are again blue and red and have the same meaning as previously stated. Butterfly dependency graphs for the directories we listed as candidates are shown in Figs. 5, 6, 15–21. Most of these figures are found in Appendix A.

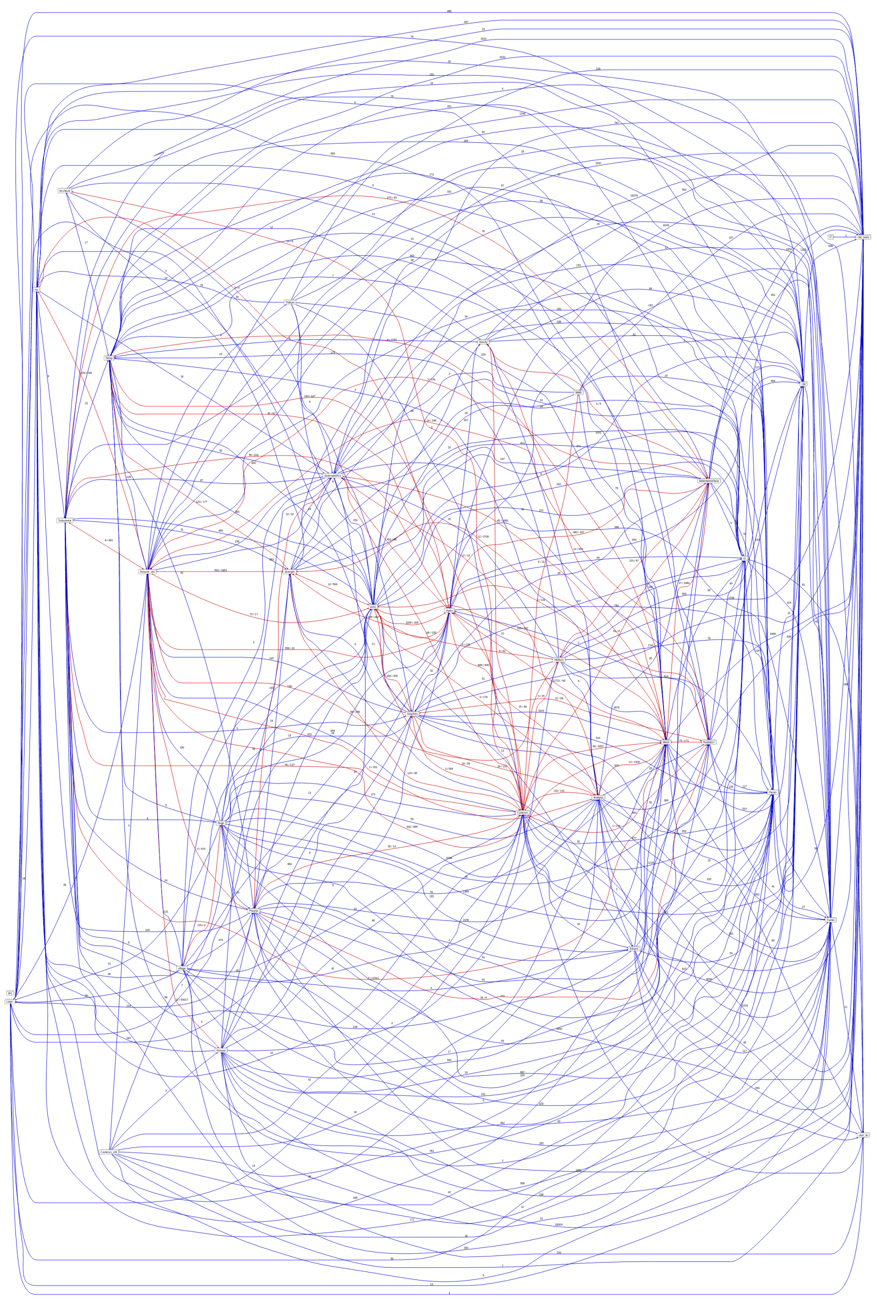


Figure 4: Directory level dependency of `Source.rh`. Blue lines indicate unidirectional dependence. Red lines indicate circular directory level dependence. An electronic version of this document allows zooming in on this figure.

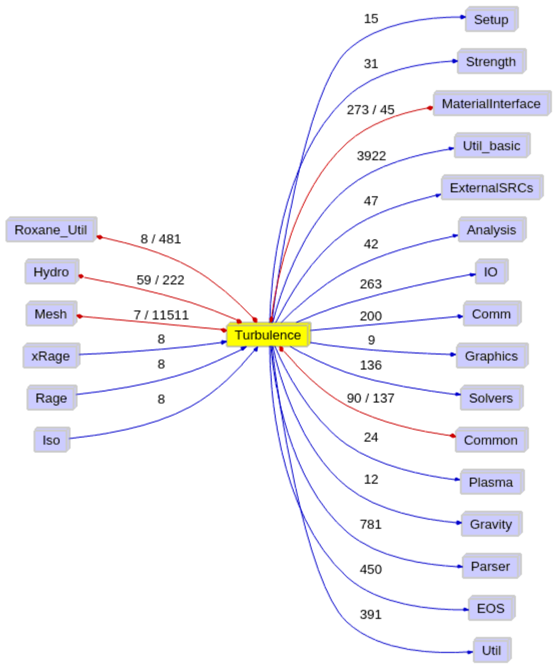


Figure 5: Butterfly dependency graph for Turbulence.

The numbers on the lines connecting two directories indicate the number of explicit dependencies of files in the first directory on files in the second directory. In the case of a circular directory dependency there are two numbers. The numbers in this case mean the same thing and the number closest to a directory is the number of dependencies of that directory on the other. The relative size of the numbers are instructive in that they can tell one what the dependency "should be." Looking at the line connecting Turbulence to Mesh in Fig. 5 one sees that Turbulence depends on Mesh several orders of magnitude more times than Mesh depends on Turbulence. This indicates that the true dependency should be of Turbulence on Mesh and the 7 dependencies in the other direction should be removed. Of course this determination of true dependencies also needs to be guided by intuition.

To quantify the interdependence of a directory on other directories we define the following metrics.

- **CC:** Coarse circular. The number of red lines connected to a directory. This number measures how many directories are connected to a directory with circular dependencies.
- **FC1:** Fine circular 1. Add up the smaller of the two numbers on every red line. If one were to remove all circular directory level dependencies at once this is the number of dependencies that would have to be removed to eliminate circular dependencies.
- **FC2:** Fine circular 2. Add up the smaller of the two numbers on all red lines *where the directory in question is the smaller of the two numbers.* The idea here is that we only consider outgoing dependencies from a directory when it should only have incoming dependencies from another directory, i.e., when the directory in question is the smaller of the two numbers on a red line. Eventually application of this process *on all directories* would remove all circular directory level dependencies.

The dependency breaking processes described in the details for **FC1** and **FC2** will both lead to eventual elimination of all circular dependencies. It is our view, however, that following the procedure outlined in **FC2** will be an easier approach.

Directory	**CC**	**FC1**	**FC2**
EOS	9	661	514
HEBurn	4	175	67
Hydro	11	1551	916
Iso	2	261	4
Plasma	4	45	19
Radiation	6	118	29
Strength	6	922	36
TNBurn	0	0	0
Turbulence	5	209	90

Table 1: Metrics for interdependence of directories.

HEBurn	19
Hydro	263
Iso	35
Plasma	30
Radiation	33
Strength	25
TNBurn	1
Turbulence	24

Table 2: Estimated test coverage.

In Table 1 the metrics are given for all of the candidate physics packages.

Next we define a metric for test coverage. In Table 2 is a rough count of the number of tests that touch the contents of each directory. EOS is not on the list because many tests beyond the targeted EOS tests use the EOS, so an accurate count is difficult to obtain. We thus exclude EOS from consideration as an initial candidate because we feel the changes to EOS will result in changes to too many tests rather than having a more targeted test suite.

We exclude TNBurn from immediate consideration because it is a new capability and there is only one test for it.

We exclude HEBurn and Turbulence directories as initial candidates based on criterion 7.

Finally, we prioritized the remaining list by our familiarity with the package, access to SMEs, and complexity. In some sense the exact ordering of the reduced set is not critical, though we feel strongly that Hydro and Radiation should be the first two worked on. If we are successful at cleaning up Hydro we will apply the methodology to all of the remaining directories in this list. Following the completion of those we can move on to other directories in the code. The prioritized list is:

1. Hydro,
2. Radiation,
3. Iso,
4. Strength, and
5. Plasma.

3 Analysis of Selected Package

In this section we describe our approach to analyzing a selected package. In some cases we will use specific examples from the Hydro package, but in general the methodology will be applicable to any package selected.

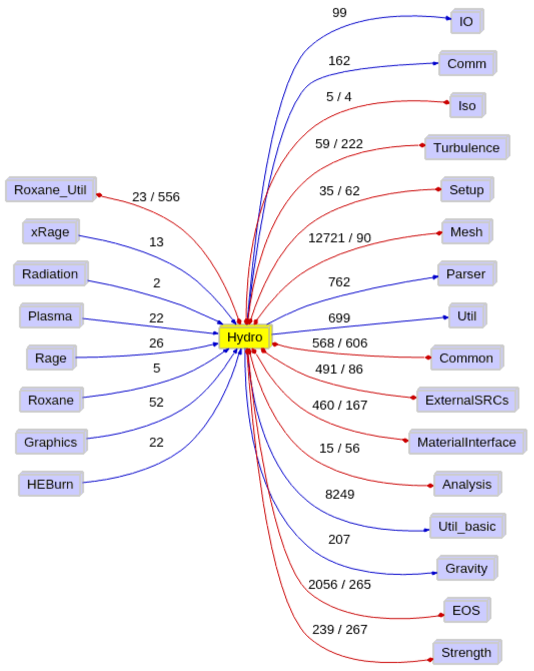


Figure 6: Butterfly dependency graph for Hydro.

The general approach is:

1. List the packages connected to the selected package and note the relationships.
2. List the packages with circular dependencies on the selected package.
3. Order the list of connected packages that match the **FC2** metric considering both the size of **FC2** and the fact that some of the connected packages may be more "package-like" than others.
4. Go through the list of connected packages and determine and document what the exact nature of the dependencies are. In particular look at both directions of the circular dependencies.
5. Based on the study of the dependencies, make a plan for the types of code transformations one wants to make. We will detail this step in Section 4.

In the following paragraphs we present the results of applying the steps listed above to the Hydro package.

1. From the butterfly diagram of Hydro (Fig. 6), the packages that have unidirectional dependence on Hydro are: Graphics, HEBurn, Plasma, Radiation, Rage, Roxane, xRage. The packages that Hydro has unidirectional dependence on are: Comm, Gravity, IO, Parser, Util, Util_basic.

2. The packages with circular dependencies on Hydro are: Analysis, Common, EOS, ExternalSRCs, Iso, MaterialInterface, Mesh, Roxane_Util, Setup, Strength, Turbulence.

3. The packages with circular dependencies on Hydro that match the **FC2** metric are: Analysis, Common, Setup, Strength, Turbulence. We order this list as follows based on both the size of **FC2** and whether the connected package is considered to be a candidate for a "real package," or is really more of a mixed bag of source files that should probably be a member of other as yet unidentified packages: Turbulence, Strength, Analysis, Setup, Common. Turbulence is listed before Strength since it has fewer dependencies and we want to start with fewer dependencies and move towards more dependencies.

4. The general approach we used to analyze the particular dependencies of two packages consisted of a series of command line queries to list modules in a directory, and then look for the use of those modules in the other directory. This was done both for the candidate package and the dependent package in question to identify exactly the nature of the circular dependence. The second part of studying the actual source dependency and noting relationships cannot be scripted; there is no way around having to consider every dependency by looking at all files identified by the first process. The results of this analysis are presented in sections 4 and 5. The cross-directory dependencies between Hydro and the five packages identified in step 3 are shown in Figures 8–12.

The analysis of Hydro and Turbulence led quickly to identification of sections of code that should likely be removed. It is important for developers working on cleanup to have a clear process for deprecating and removing code. See Section 4 for details of our proposal for this process.

4 Deferred Dependency Removal and Deprecation

As was described in § 3, dependencies between Hydro and packages that are positioned lower in the levelization diagram will be addressed when those lower packages are subjected to this process. For example, we can create a conceptually clear Hydro package interface that is exposed to the packages that depend on Hydro without first addressing, e.g., Hydro's dependence on `seteng` in Common.

Also, we will not clean up a dependency when it exists to support functionality that is a candidate for deprecation and eventual removal. The determination of what code and functionality should be deprecated and removed is clearly outside the authority of the authors of this memo. We propose a decision making process depicted in Figure 7. The steps in this decision making process begin with a specific package or section of code and are described below. The actors at each step are listed in italics.

1. *Developer* Prior to committing time to disentangling and cleaning up code within a package, the developer will first ask whether the code is in use. This initial step can include examining run logs and validation suites. If the developer determines that the answer is yes, as will often be the case, the developer should proceed to step 7. If the developer cannot clearly see that the code is used, the process moves to step 2.

2. *SME* The SME responsible for the functionality in question has the authority to decide whether a section of code and the associated functionality should be deprecated. If the SME feels that the code should not be deprecated, the process moves to step 7; otherwise the process moves to step 3.

3. *SME, Management, Users* Here the SME and management will engage with the users to determine if the code is used in practice. If it is not, the process moves to step 8. Otherwise we proceed to step 4.

4. *SME, Users* The SME and users will identify the way in which the code is being used. We believe that this will consist of the SME talking to users about the way they are using the functionality associated with the code in question. The process now moves to step 5.

5. *SME, Management, Users* At this point a determination will be made as to whether the code in question must be kept. Considerations might include whether the SME feels that the same capability can be supported with different functionality, and the degree to which the existing functionality is critical for the user to perform her or his work. If it is determined that the code must be kept the process moves to step 6, otherwise the process moves to step 8.

6. *Developer, SME* The code to be kept and the associated usage will be documented, and the process moves to step 7.

7. *Developer* The developer will determine if the code is covered by the test suite. If it is, the process moves to step 10.

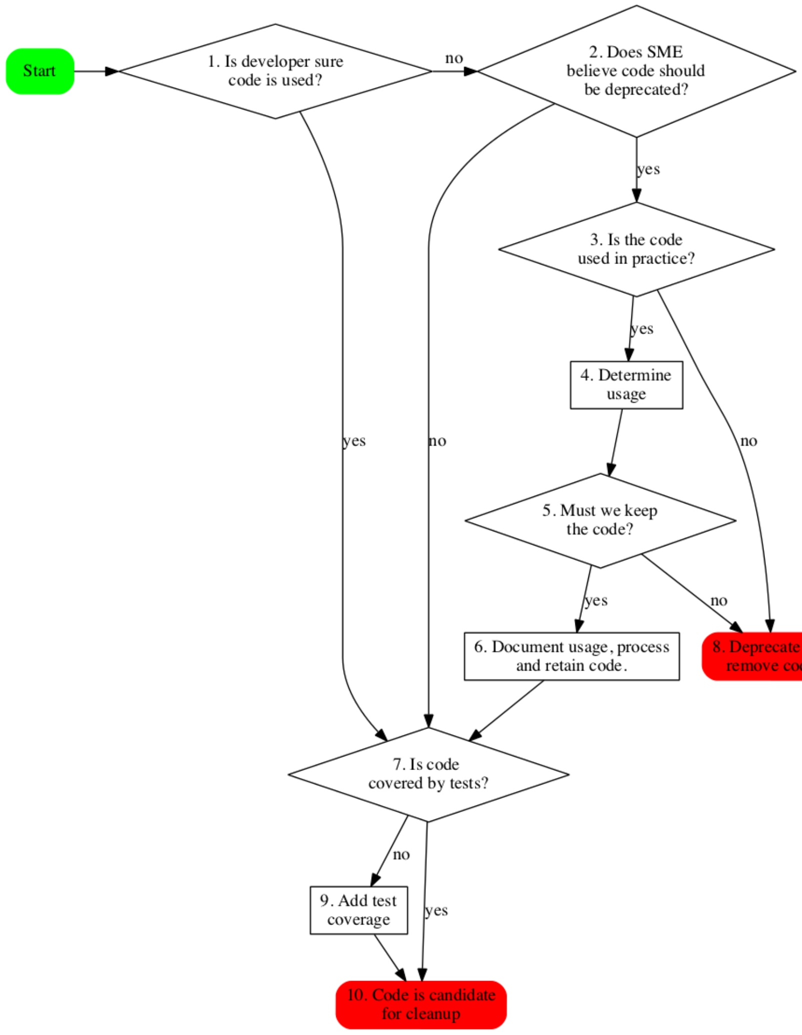

Start
1. Is developer sure code is used?
no
2. Does SME believe code should be deprecated?
yes
3. Is the code used in practice?
yes
4. Determine usage
yes
no
no
5. Must we keep the code?
yes
no
6. Document usage, process and retain code.
8. Deprecate and remove code
7. Is code covered by tests?
no
9. Add test coverage
yes
10. Code is candidate for cleanup

Figure 7: A flow chart for the process by which code is evaluated for removal or clean up.

8. *Developer, SME* The code and associated functionality will be marked as deprecated[2] in the next release of Rage. No further support will go into deprecated code. In the following release the deprecated code will be removed. This is one of the end points of this process.

9. *Developer, SME* The developer and SME will develop and add a test that exercises the code in question, and the process moves to step 10.

10. *Developer* The developer begins the clean up process for this section of code. This is the other end point of this process.

There are several features of this process that are worth pointing out. First, this process ensures that retained code is covered by the test suite. Second, we expect that in the majority of the cases, the path through this process will be very short—the developer will immediately make the determination that the code is in use, and will make sure that the code is covered by the test suite and proceed. Third, the determination of whether code should be deprecated requires active user participation—users will not have code or functionality unexpectedly vanish. Fourth, the determination of whether code, which the SME feels should be deprecated, should actually be deprecated does *not* require the involvement of the developer. We can envision that this process could require resolving a disagreement between the users and SMEs. If this is the case, the developer is free to work on other efforts.

5 Anticipated Transformations for Selected Package

5.1 Characterizing dependencies

There are two broad classes of circular relationships between Hydro and other directories:

- *intrinsically-dependent*: subroutines and references to data in modules in the two directories will be interleaved in the call stack at some point during execution; and

- *coincidentally-dependent*: subroutines and data in modules in the two directories will not be interleaved in the call stack during execution; the circular relationship at the directory level consists of a set of one-directional dependencies at the module level.

An intrinsically-dependent relationship between directories does not imply that the relationship is somehow fundamental to the underlying algorithm. Instead, the dependence is "intrinsic" to the existing implementation; disentangling the relationship will require changes to code, not just moving or renaming files or directories.

The dependencies between Hydro and Strength (Figure 8), between Hydro and Turbulence (Figure 9), and between Hydro and Setup (Figure 10) are typical examples of intrinsically-dependent directories. Modules in Hydro (`hydro_1_module`, `hydro_3_module`, `hydro_14_module`, `hydro_mp_module`, `hydro_muscl_module`, and the top-level `hydro_module`) use subroutines in `strength_module`, which calls into `strength_1_module` if that strength model is active, which in turn uses data from Hydro's `mesh_state_hydro_module`. In the other direction, `hydro_14_module` uses data from `strength_data_module`, so modules in each directory can operate directly on data that nominally "belong" to modules in the other. The relationship between Hydro and Turbulence is simpler, though similar; several Hydro modules call subroutines in `mix_module`, while several Turbulence modules (including `mix_module`) directly use parameters and properties from `hydro_params`, `hydro_prop_module`, and `interface_vars_module`. The relationship between Hydro and Setup is another variation on the pattern: `gfm_module` and `mhd_module` operate directly on data from Setup's `define_regions` and `merge_data`, while `regions_module` and `fndzzz_module` in Setup access parameters and properties in `hydro_params` and `hydro_prop_module`; `regions_module` also invokes subroutines in

[2] "Marked as deprecated" should consist, at minimum, of notification in the release notes for the release in which deprecation occurs. It could also include adding additional logging for deprecated features to collect uses of such features ahead of their removal.

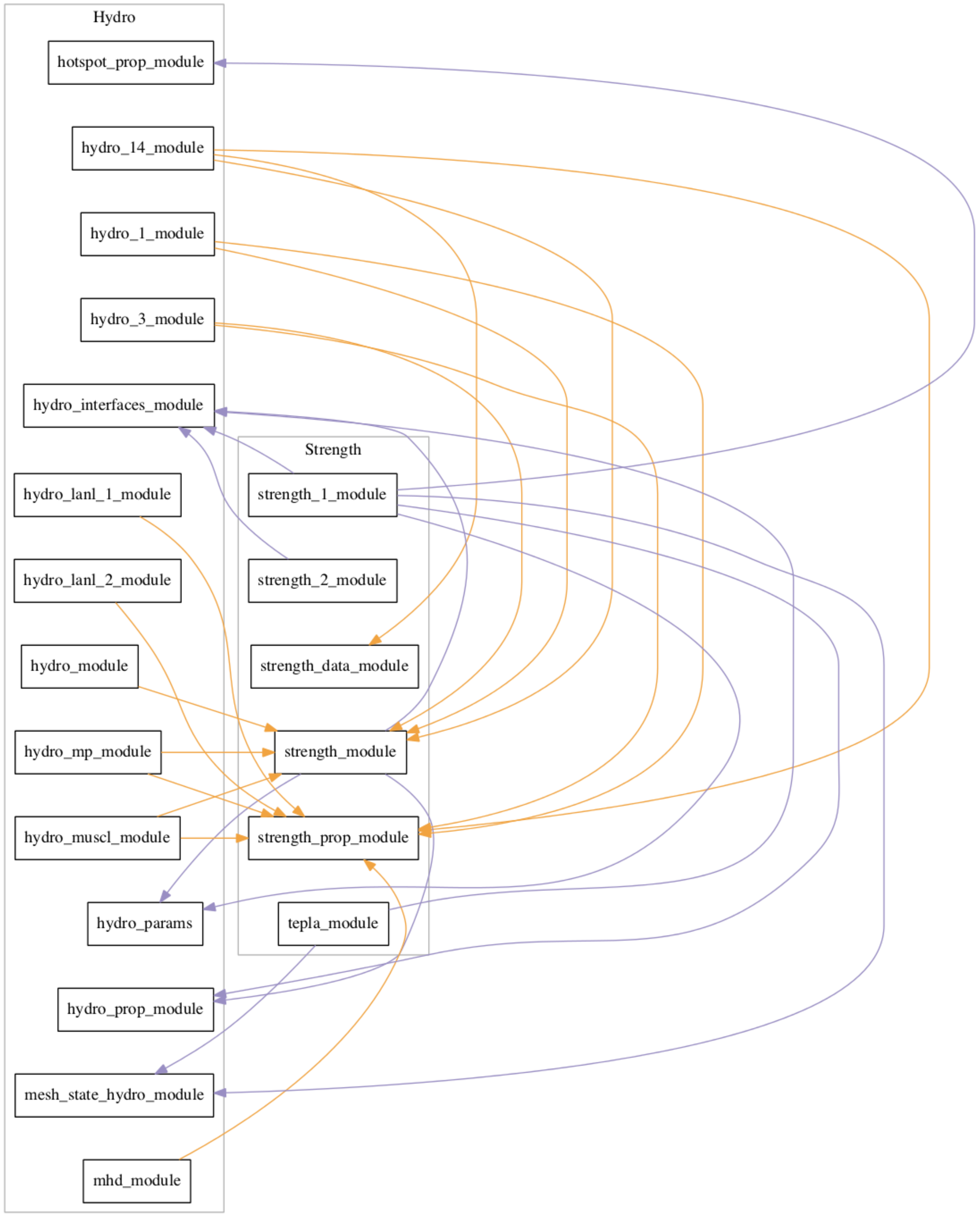


Figure 8: Cross-directory dependencies between Hydro and Strength, as of SVN rev 18865.

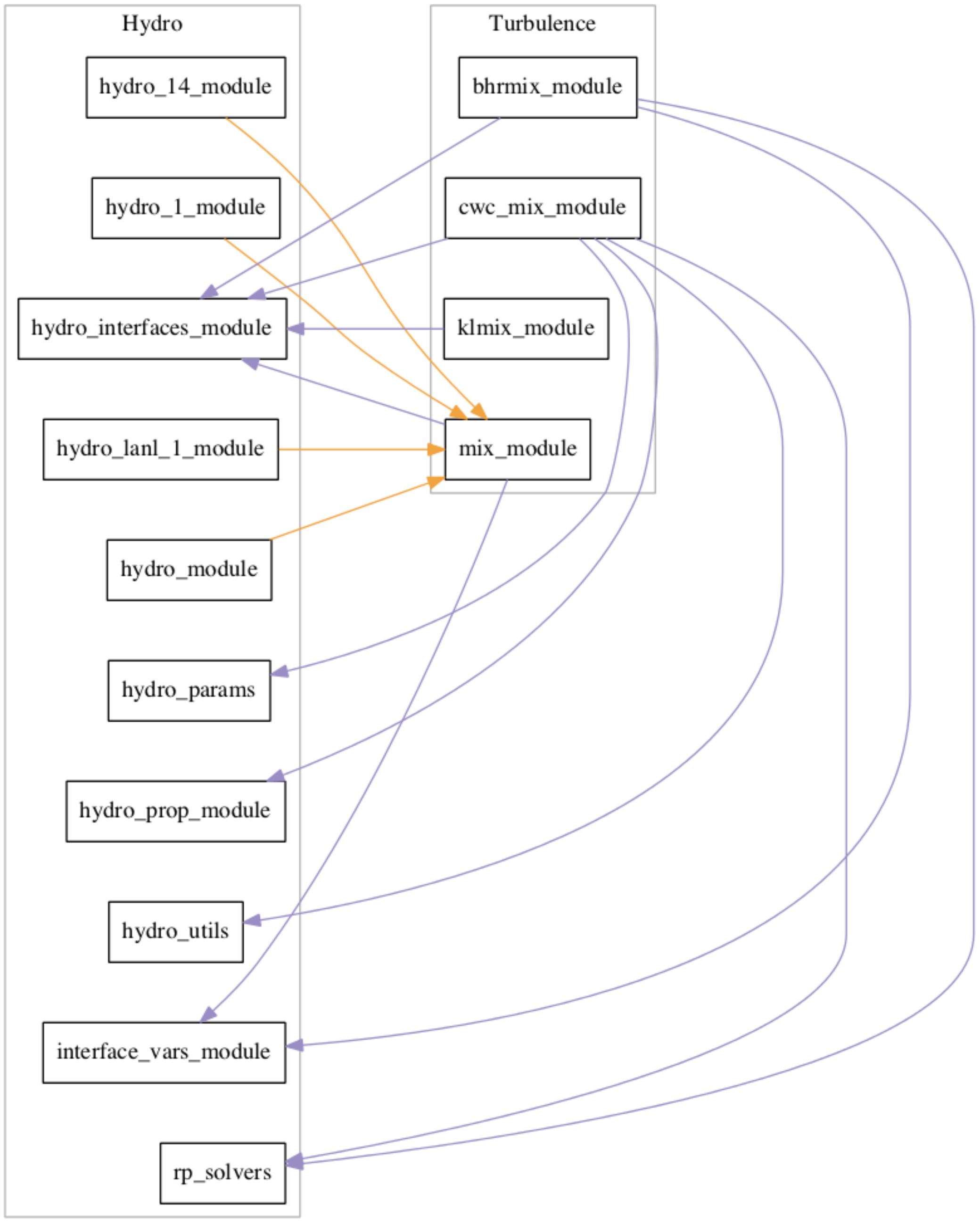


Figure 9: Cross-directory dependencies between Hydro and Turbulence, as of SVN rev 18865.

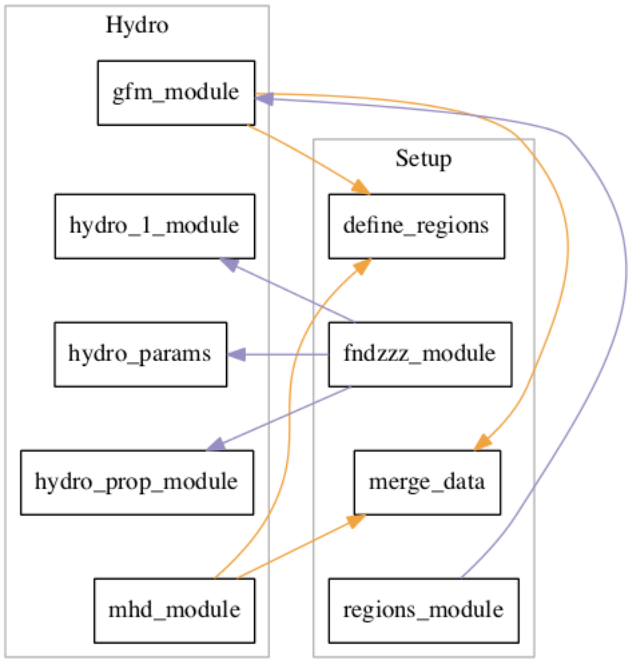


Figure 10: Cross-directory dependencies between Hydro and Setup, as of SVN rev 18865.

`gfm_module` to complete GFM-specific setup tasks and to trigger GFM-specific consistency checks. This recurring pattern, where modules rely on `use` statements to operate directly on data owned by other modules without their knowledge, results in separation between data flow and control flow, which increases the difficulty of reasoning about the overall state of data at any given point in the code.

The dependency between Hydro and Analysis (Figure 11) is an example of a coincidentally-dependent relationship. Most of the modules in these directories are absent from the cross-directory dependency diagrams, because they don't depend on any modules in the other directory in the diagram. While the directory-level relationship between Hydro and Analysis looks bi-directional, the module-level interactions only extend one level deep: `hydro_module` invokes subroutines in `ie_edit_module` and uses a logical parameter in `ie_data_module`, while `editcycle_module` in Analysis uses parameters and properties from `hydro_params`, `hydro_prop_module` and `interface_vars_module`.

The dependencies between Hydro and Common (Figure 12) combine features of both types of relationships. `test_hydro_module` in Hydro relies on several modules in Common (`rt_module`, `sedov_module`, `stube_module`, and `test_answers_module`) to support construction of hydrodynamic test problems. Two modules in Hydro (`hydro_1_module` and `hydro_14_module`) refer to `tinyvel` in `mesh_state_tstep_module`; several other modules depend on `cdt` and `tsteppu t` in `tstep_module`. Many modules in Hydro use `seteng` and `engchk`, from `energy_module` in Common. The only common feature among these groups of modules in Common is that Hydro modules are not the only modules that depend on them, so they've been pushed down the hierarchy and collected in a single directory. At the same time, many modules in Common, including `energy_module`, depend on properties and data in `hydro_params`, `hydro_prop_module` and `mesh_state_hydro_module`.

A few cross-directory dependencies are trivial to eliminate. For example, `mesh_state_radiation_module` appears to depend on `hydro_prop_module` (Figure 13), but the entity specified in the only `use` statement, `hdp`, is not actually used in the associated subroutine. Simply deleting the `use` statement eliminates the dependency.

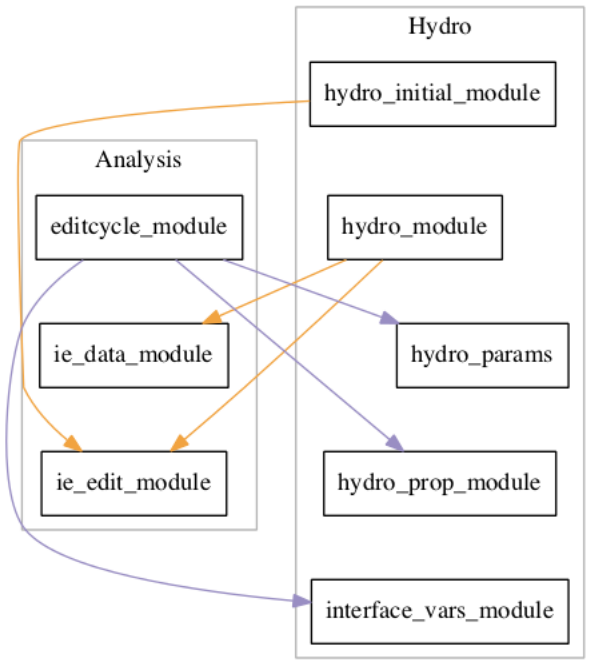


Figure 11: Cross-directory dependencies between Hydro and Analysis, as of SVN rev 18865.

The examination of dependencies above hasn't considered extras chains, but those chains are an important expression of cross-package relationships, and transforming them is part of this disentangling task. Cross-directory dependencies through extras chains are not captured in the figures above because the entry point for most chains (Common/module_code_extras_1.f90) is not a module, so callers don't refer to it via `use` statements. Fortran interfaces do exist for many of the extras subroutines, but the interface to a given extras chain often lives in a module in the same directory as the subroutine that calls the chain. For example, hydro_extras_int.f90 in Hydro provides an interface for the `hydro_advect_extras` chain, so callers in Hydro appear to have a Hydro-internal dependence on `hydro_extras_interface_module`, not a cross-directory dependence on the contents of module_code_extras_1.f90 in Common. Nevertheless, the extras chains generally represent variations on the intrinsically-dependent class of cross-directory relationships described above, and they can be transformed using the same process described below.

5.2 Transforming code to eliminate dependencies

As was described in § 4, determining whether any given capability (large or small) can be deprecated and removed is an important first step in reducing the total volume of code that needs to be modularized. Hydro's dependence on Turbulence highlights the value of identifying and removing deprecated code; deprecating and removing the "VOF mix" capability in `mix_module` would eliminate all dependencies on modules in Turbulence by modules in Hydro. Extending this process to entire hydro versions would eliminate even more cross-directory dependencies and would help focus refactoring effort.

For capabilities that will be retained, the transformation described by Andy Nelson in his "EAP Refactoring Overview" (October 31, 2013) and summarized in § 1 can help to establish clear interfaces between packages and to clarify data flow. At a high level, the refactoring process should consolidate the expression of the protocol for a given phase of a calculation (e.g., "hydro") in a high-level driver for that phase and should establish an interface to each involved package that allows the driver to interleave calls to different packages as required by the protocol. Individual packages should not rely on `use` statements to access data

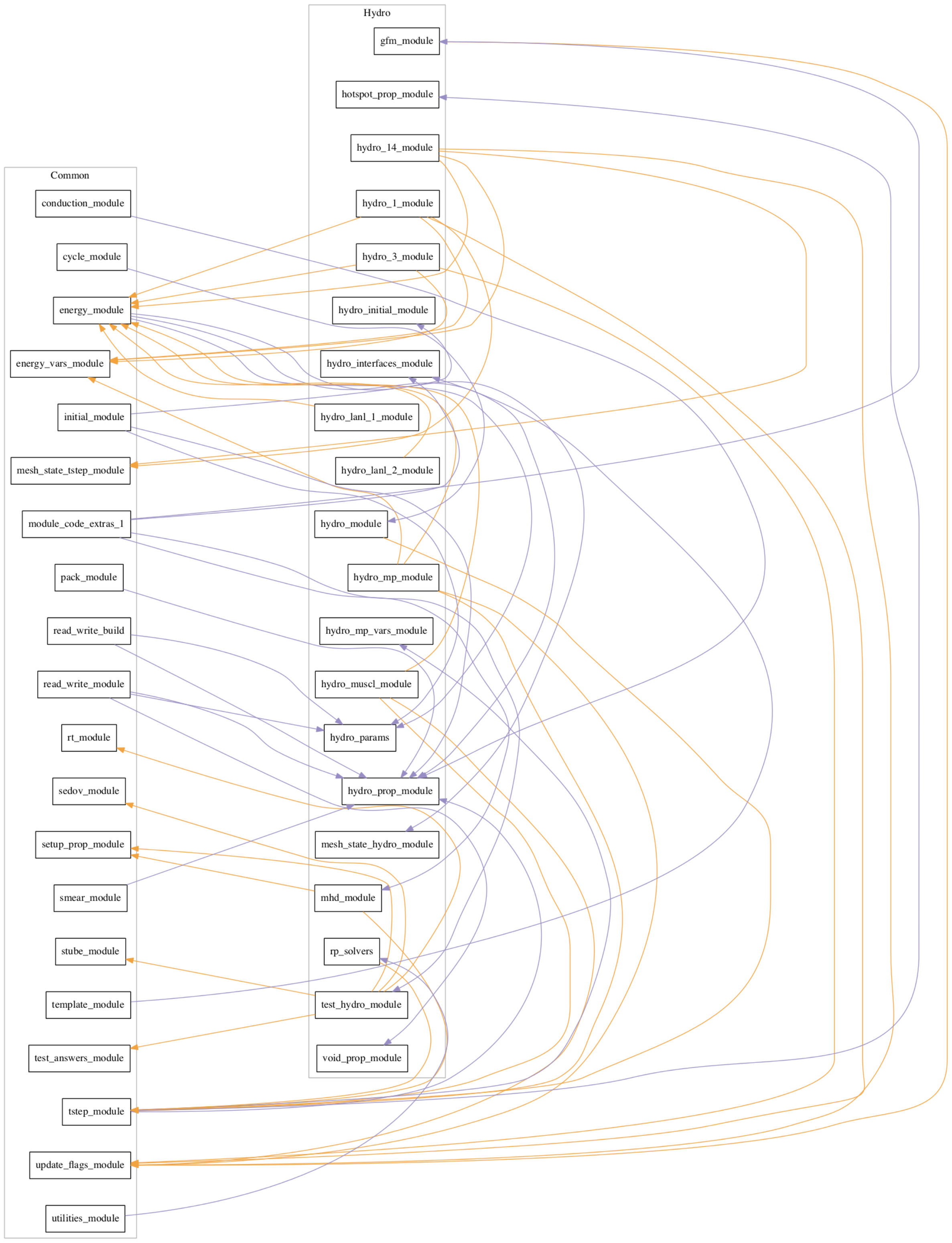


Figure 12: Cross-directory dependencies between Hydro and Common, as of SVN rev 18865.

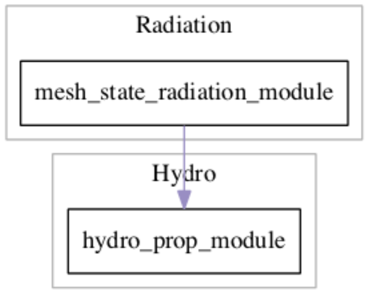


Figure 13: Cross-directory dependencies between Hydro and Radiation, as of SVN rev 18865.

from other packages; instead, they should provide interfaces to the caller that express their specific data needs and expect the caller to provide that data to them.

Figure 14 shows the call graph from the top-level `hydro` subroutine to the routines in Strength, Analysis, Turbulence, and most of the routines in Common that are called by modules in the Hydro directory. Transforming the code as described in § 1 will involve:

1. pushing the `hydro` subroutine up into a new directory (tentatively, "hydro_phase", a peer of future driver packages for other phases of the calculation) to identify it as the driver for the hydro phase;
2. consolidating the expression of the protocol for the hydro phase by splitting large subroutines and flattening the levels evident in Figure 14 until the sub-steps in the phase (i.e., most of the major routines in that call graph) are all invoked directly by the driver, in the correct order;
3. moving data `use` statements up the call hierarchy into the top-level driver, and replacing them with explicit arguments passed to lower-level subroutines.

After that transformation, the top-level driver subroutine will depend on existing modules in lower-level directories, but cross-directory dependencies among those lower-level directories will have been eliminated, and the overall relationship among the directories affected by the change will form a directed acyclic graph. The same transformation can be applied to extras chains to consolidate the subroutines invoked in the chain in a driver for the overall phase.

6 Completion Criteria and Next Steps

The completion criteria for the work proposed in this memo is as follows.

1. Elimination of circular dependencies between Hydro and capabilities at the same or higher levels in the Physical Model.
2. Creation of interfaces with explicit treatment of required data between the Hydro driver and the components of the package.
3. Coverage of the new interfaces by tests.
4. A documented methodology for cleaning up and packagizing a physics capability in xRage that can be disseminated to EAP SMEs.

What we have presented in this document represents to us the minimum amount of preparation needed before beginning the effort to clean up the packages described. We will carefully document our transformations and write up a detailed report of the work after it is completed. In our view, then, it is the current document *together with* a follow on document that we hope will serve as a blueprint that can be shared with SMEs for more broad application to EAP codes.

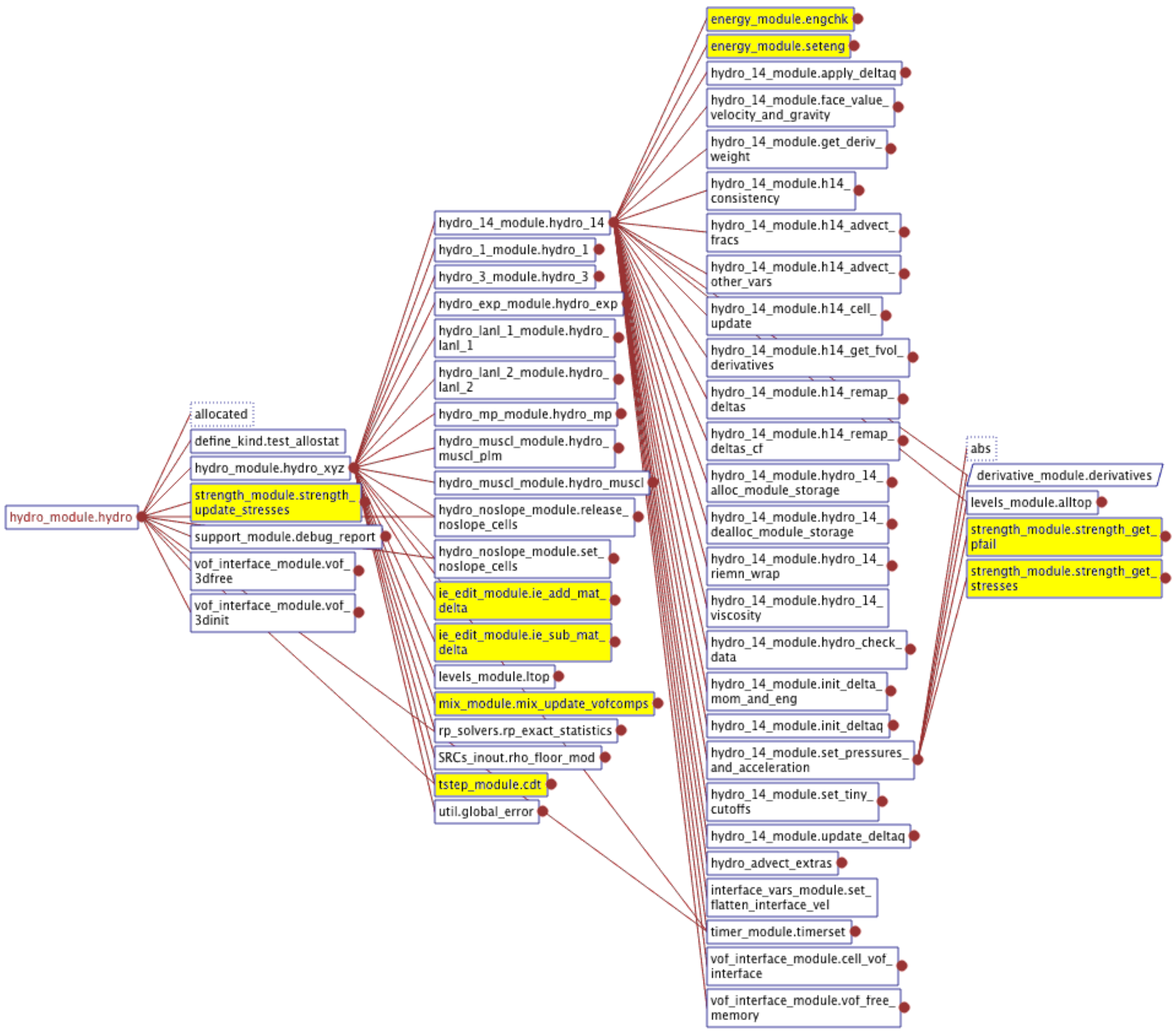

energy_module.engchk
energy_module.seteng
hydro_14_module.apply_deltaq
hydro_14_module.face_value_velocity_and_gravity
hydro_14_module.get_deriv_weight
hydro_14_module.h14_consistency
hydro_14_module.hydro_14
hydro_14_module.h14_advect_fracs
hydro_1_module.hydro_1
hydro_3_module.hydro_3
hydro_14_module.h14_advect_other_vars
hydro_exp_module.hydro_exp
hydro_14_module.h14_cell_update
hydro_lanl_1_module.hydro_lanl_1
hydro_14_module.h14_get_fvol_derivatives
hydro_lanl_2_module.hydro_lanl_2
hydro_14_module.h14_remap_deltas
allocated
hydro_mp_module.hydro_mp
define_kind.test_allostat
hydro_14_module.h14_remap_deltas_cf
abs
hydro_muscl_module.hydro_muscl_plm
hydro_module.hydro_xyz
derivative_module.derivatives
hydro_14_module.hydro_14_alloc_module_storage
hydro_muscl_module.hydro_muscl
strength_module.strength_update_stresses
levels_module.alltop
hydro_module.hydro
hydro_noslope_module.release_noslope_cells
hydro_14_module.hydro_14_dealloc_module_storage
strength_module.strength_get_pfail
support_module.debug_report
hydro_noslope_module.set_noslope_cells
hydro_14_module.hydro_14_riemn_wrap
vof_interface_module.vof_3dfree
strength_module.strength_get_stresses
ie_edit_module.ie_add_mat_delta
hydro_14_module.hydro_14_viscosity
vof_interface_module.vof_3dinit
ie_edit_module.ie_sub_mat_delta
hydro_14_module.hydro_check_data
levels_module.ltop
hydro_14_module.init_delta_mom_and_eng
mix_module.mix_update_vofcomps
hydro_14_module.init_deltaq
rp_solvers.rp_exact_statistics
hydro_14_module.set_pressures_and_acceleration
SRCs_inout.rho_floor_mod
tstep_module.cdt
hydro_14_module.set_tiny_cutoffs
util.global_error
hydro_14_module.update_deltaq
hydro_advect_extras
interface_vars_module.set_flatten_interface_vel
timer_module.timerset
vof_interface_module.cell_vof_interface
vof_interface_module.vof_free_memory

Figure 14: The call graph from `hydro` to `strength_update_stresses` in Strength, one level down; to `ie_add_mat_delta` and `ie_sub_mat_delta` in Analysis, to `mix_update_vofcomps` in Turbulence, and to `cdt` in Common, two levels down; to `engchk` and `seteng` in Common, three levels down, and to `strength_get_pfail` and `strength_get_stresses` in Strength, four levels down, as of SVN rev 18865. Our goal is to convert this call graph into an instance of the pattern in Figure 3b.

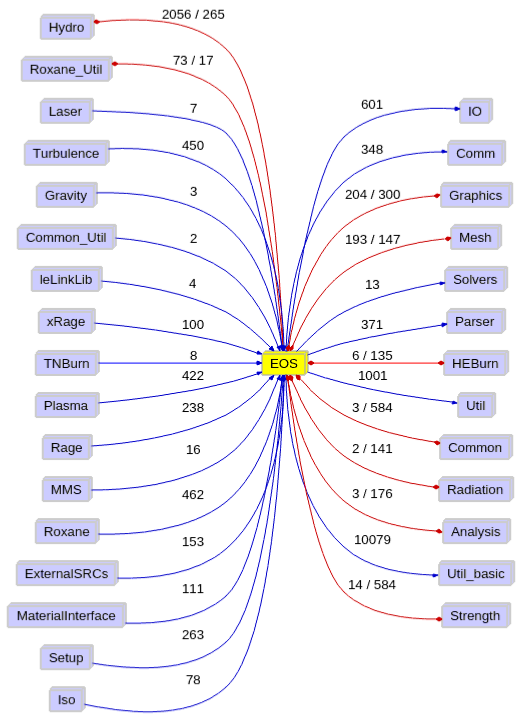


Figure 15: Butterfly dependency graph for EOS.

A Butterfly Diagrams

Butterfly diagrams for packages initally considered.

TK:tk
Distribution:
Aimee Hungerford (XTD–IDA)
Chuck Wingate (XCP–2)
Mike Steinkamp (XCP–2)
Sriram Swaminarayan (CCS–7)

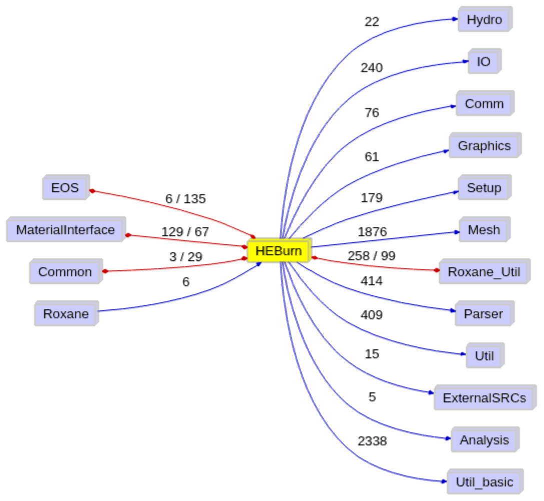


Figure 16: Butterfly dependency graph for HEBurn.

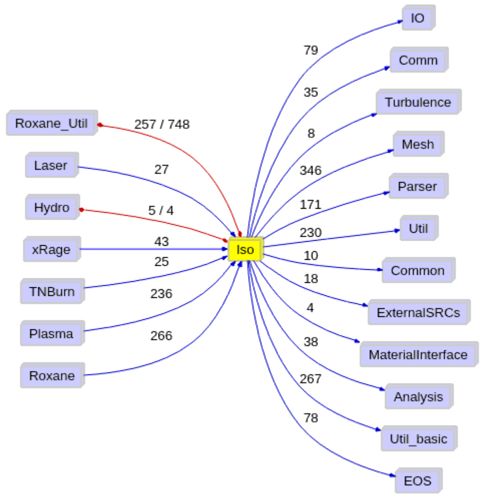


Figure 17: Butterfly dependency graph for Iso.

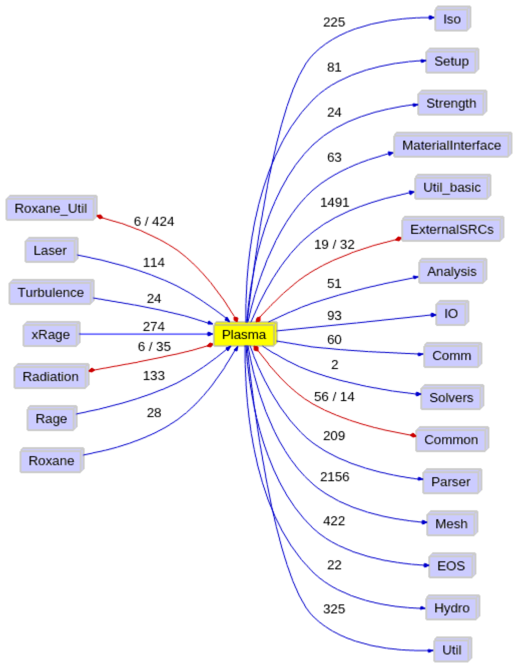


Figure 18: Butterfly dependency graph for Plasma.

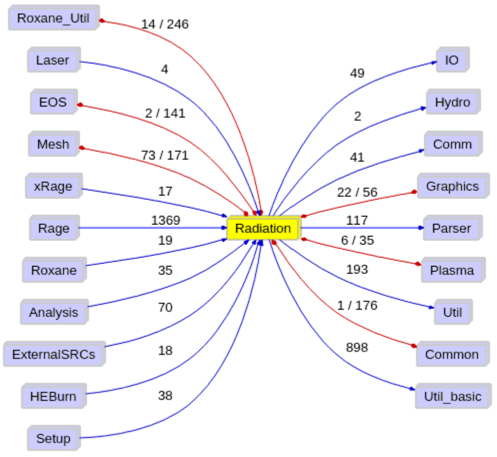


Figure 19: Butterfly dependency graph for Radiation.

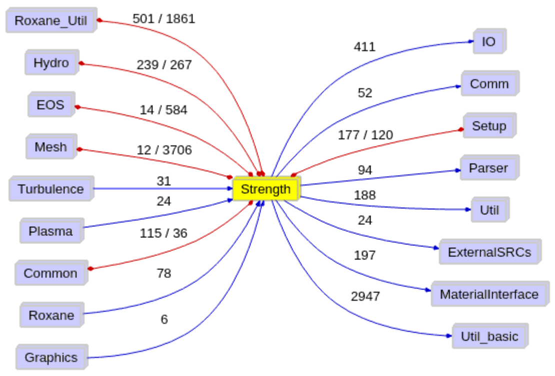


Figure 20: Butterfly dependency graph for Strength.

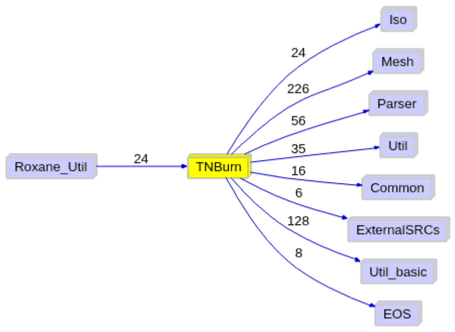


Figure 21: Butterfly dependency graph for TNBurn.